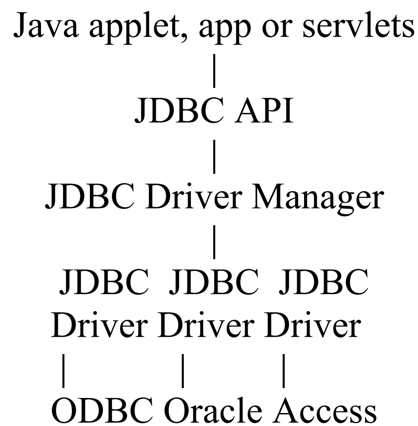
JDBC

Stands for Java Database Connectivity, is an API specification that defines the following:

1. How to interact with database/data-source from Java applets, apps, servlets
2. How to use JDBC drivers
3. How to write JDBC drivers

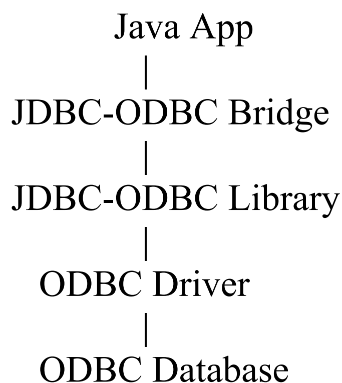
The main point of JDBC is database interoperability: By using JDBC API to database access you can change the underlaying database driver(or engine) without modifying the application.

You write standard JDBC API application and plug in the appropriate JDBC driver for the database the you want to use.



JDBC Driver Types : Four types of JDBC drivers.

Type 1: The JDBC-ODBC Bridge



JDBC-ODBC Bridge is provided as a part of JDK higher than 1.1 bridge is part of sun.jdbc.odbc package

Uses:

1. Quick system prototype

2. Three-tier database design
3. Database systems that provide only ODBC driver and not JDBC
4. Low-cost database solution where you have an ODBC driver

Limitations:

1. Bridge was developed as a prototyping and marketing tool and one should not use it for mission-critical application if native JDBC driver is available.
2. Bridge uses C language code so can not be used in untrusted applets. So this depends on operating systems.
3. Since Bridge uses existing ODBC driver any bug in the driver would be carried-forward
4. If a thing that is not possible in ODBC driver that features will not be available using bridge

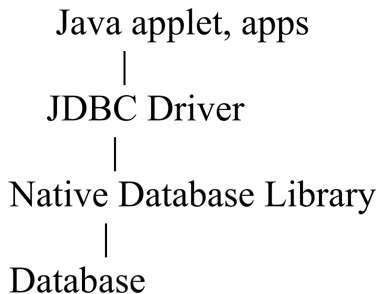
Type 2: Java to NATIVE API

It uses local native library provided by a vendor to communicate directly to the database.

This driver has many same restrictions as bridge because it uses native library but they are faster than bridge.

It can not be used in untrusted applets.

This native library must be installed on each machine which uses the driver.

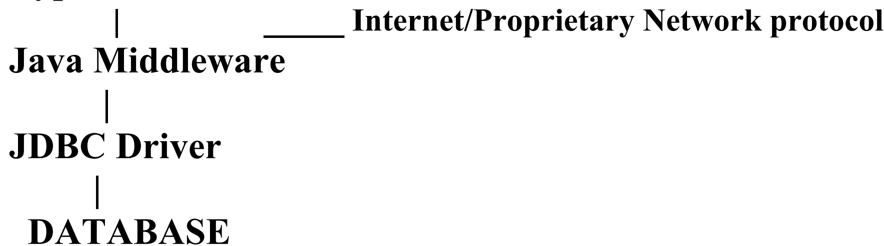


Low-cost database solution where we are already using a major database system that provides a type 2 driver with their database.

Type 3: Java to Proprietary Network Protocol

Java application, applet, servlets

Type 3 JDBC driver



- This type of JDBC driver is by far most flexible.
- Used in n-tier application and can be deployed over Internet.
- Type 3 driver are pure Java and communicate with some type of middle tier via a proprietary network protocol created by the driver vendor.
- This middle tier will most likely reside on a Web or database server and, in turn, communicate with the database.

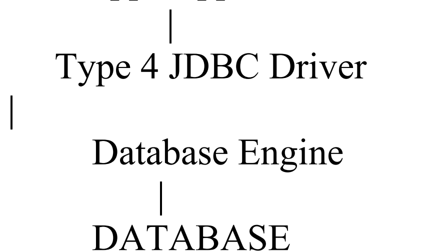
Uses:

- Web-based applet that do not require any pre-installation or configuration
- Secure systems where the database producer will be protected behind middle tier
- Flexible solution where many different database products in uses
- Clients require a small footprint-size of type 3 drivers is much smaller than all other types.

Type 4: Java to Native database protoco*l*

- Type 4 drivers are pure Java drivers that communicate directly wih database engine via its native protocol.
- This driver can be deployed over Internet.
- This driver can give higher performance as compared to first two.

Java Apps, applets, Servlets



Uses:

Performance-critical apps

In environments where only one database product is in use

Web-deployed applets

The JDBC program flow

1. Establish a connection to the database
2. Execute a SQL statement
3. Process the result
4. Disconnect from the database

Loading Drivers

Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Your driver documentation will give you the class name to use. For instance, if the class name is `jdbc.DriverXYZ` , you would load the driver with the following line of code:

```
Class.forName("jdbc.DriverXYZ");
```

1. Establish a connection to the database
   JDBC connections are specified by URL string which has format:
   Jdbc:<subprotocol>:URL
   e.g. jdbc:odbc:dsnName

Connection con =
DriverManager.getConnection("jdbc:odbc:MsAccess");

If we are using the JDBC-ODBC Bridge driver, the JDBC URL will start with `jdbc:odbc:` . The rest of the URL is generally your data source name or database system.

Connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS.

2. Execute SQL Statement
Statement – class
Prepared Statement – to create object using SQL Statement
Callable Statement – to call backend procedure

```
Public class simple query
{
     pubic static void main(String args[])
      {
           String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
           String connectionURL = "jdbc:odbc:AccessDb";
           Class.forName(driverName).newInstance();
           Connection con = DriverManager.getConnection(connectionURL);
}
}
```

<u>Establishing a Database Connection: *java.sql package*</u>

Public static synchronized Connection getConnection(string url, String user, String password) throws SQLException
Con =
DriverManager.getConnection("jdbc:odbc:AccessDSN", "uid", "pwd")

<u>Interacting with Database using SQL:</u>

To execute SQL command using a JDBC connection, we must create a JDBC Statement object.
By   Connnection.createStatement() method.

Table : Stud_id integer, Stud_Name;

```
Import java.sql.*;
public class FirstSQLApp {

public static void main(String args[]) {
Connection con = null;
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Con= DriverManager.getConnection("jdbc.odbc.AccDSN","","");
```

```
Statement s1 = con.createStatement();
Statement.excuteUpdate("Create Table Stud " +
           "(stud_id INTEGER, stud_name CHAR(30))");
}
catch(SQLException e1)
{
        System.err.println(e1.getMessage());
}
catch(Exception e2)
{
        System.err.print(e1.getMessage());
}
finally
{
        try {
        if(con!=null) {
        con.Close(); }
        }
catch (Exception e) { } }
```

Inserting Data into Table: Insert SQL Statement

```
s1.executeUpdate("insert into Stud " + "values (1,'stud1')");
```

Selecting Data from Table: SQL Select

JDBC ResultSet object is used as client-side cursor to select data.

```
try {
ResultSet rs = statement.executeQuery("Select * from STUD");

while(rs.next())
{
        System.out.println("Stud Id = " + rs.getString("stud_id"));
        System.out.println("Stud Id = " + rs.getString("name"));
}
rs.close();
}
```

getXXX methods

rs.getFloat(),
rs.getInt() …

Updating Tables

```
Statement.executeUpdate("UPDate stud " +
        "SET name = 'stud2'" +
        "Where stud_id = 1");
```

Deleting from Table

```
        Statement.executeUpdate("DELETE from stud " +
                "Where stud_id = 2");
```

Types of Statements:

1. PreparedStatements - to create parameterised query
2. Callable statements – to execute stored procedure

Prepared Statements

`PreparedStatement` object contains not just an SQL statement, but an SQL statement that has been precompiled.
We can use them for, which take parameters and the same SQL statement is being executed again and again.

Creating a PreparedStatement Object

```
PreparedStatement s2 =
con.prepareStatement( "UPDATE  Stud SET name = ? Where  stud_id =?");

updateSales.setString(1, "stud3");
updateSales.setInt(2, 1);
s2.executeUpdate();
```

Return Value for PreparedStatement Object

```
int n = s2.executeUpdate();
```

Stored Procedures

Calling a Stored Procedure from JDBC

```
CallableStatement cs = con.prepareCall("{call SHOW_STUDLIST}");
ResultSet rs = cs.executeQuery();
```

CallableStatement is subclass of PreparedStatement so callable statement can also take input parameter in the same way.

Transactions

A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

```
con.setAutoCommit(false);
```

Committing a Transaction

```
con.setAutoCommit(false);
…….
PreparedStatement/Statement/CallableStatement executeUpdate
…….
con.commit();
con.setAutoCommit(true);
```

RollingBack a Transaction

```
Con.RollBack();
```

Transaction Isolation Level:

To avoid conflicts during a transaction, a DBMS will use locks, mechanisms for blocking access by others to the data that is being accessed by the transaction.

Default Isolation Level:
`TRANSACTION_READ_COMMITTED`:
DBMS will not allow dirty reads to occur

Method:
GetTransactionIsolation - to get Isolation Level of DBMS
SetTransactionIsolation - to set Isolation Level of DBMS